UNIVERSITY OF WATERLOO
Cheriton School of Computer Science

CS 458/658 **Computer Security and Privacy** **Fall 2013**
Douglas Stinson **Ryan Henry**

ASSIGNMENT 3

Assignment due date: **Friday, November 29th, 2013 @ 6:00 pm**

TA: Jalaj Upadhyay <jalaj.upadhyay@uwaterloo.ca>
Office hours: Tuesdays, 3:30 – 4:30 in DC 2102
Total marks: 50

Written Response Questions [30 marks]

Note: For written questions, please be sure to use complete, grammatically correct sentences where appropriate. You will be marked on the presentation and clarity of your answers as well as the content.

1. [8 marks total] **Anonymous communications systems.**
   Peer-to-peer systems often rely on the existence of many independent (i.e., non-colluding) remote entities to mitigate threats from hostile peers. The "Sybil" attack (`http://www.cs.rice.edu/Conferences/IPTPS02/101.pdf` — not a required reading) is a general attack on such systems wherein the attacker accumulates sufficiently many distinct "identities" in the system to violate the necessary conditions for security. Tor and other "low-latency" anonymous communication systems are inherently susceptible to Sybil attacks: An adversary who controls some fraction f of the Tor network's capacity can deanonymize about a fraction f^2 of all connections through Tor.

   (a) [2 marks] Explain how an adversary that controls the first and last hop in a Tor circuit might be able to deanonymize that circuit.

   (b) [2 marks] Anyone (including you!) can run a Tor relay. Tor lets you configure your own relay as an "exit relay", which can act as the last hop in a circuit and relay traffic to the public Internet, or as a "non-exit relay" that only routes traffic between other relays and Tor clients. Explain the reason for this configuration option; that is, why might a relay operator prefer to run a non-exiting relay?

   (c) [2 marks] In part (b), we learned that there may be good reasons to not run an exit relay. As it turns out, running an exit node can actually be of interest to nefarious entities (or to researchers). Why?

(d) [2 marks] So far, we have considered low-latency anonymous communications systems; for the last part of this question we switch our focus to high-latency anonymous communications systems. Consider an anonymous remailer network in which mix nodes wait until they have received a batch of N messages, at which point they randomly permute the order of the messages, decrypt one layer of encryption from each message, and then forward each message to its next hop. How might an adversary who is capable of observing the (encrypted) packets flowing to and from a particular mix node correlate incoming/outgoing messages to/from that mix node?

2. [8 marks total] **RSA Encryption.**
In the lectures, you were told that public key encryption is slower in comparison to symmetric key encryption. This is true; however, there are ways to speed up some encryption schemes. The objective of this exercise is to show a way to speed up the RSA algorithm. We use the following notation: if $a - b$ is a multiple of a number c, we write $a \equiv b \mod c$.

In RSA, the public key is (n, e), where $n = pq$ for large primes p and q, and e is an integer.

The private key is (p, q, d) for d such that $de \equiv 1 \mod (p-1)(q-1)$. The encryption of a message m is $c = m^e \mod n$ and the decryption is $m \equiv c^d \mod n$.

(a) [2 marks] Implement this algorithm in MAPLE (or any programming language of your choice) and compare the running time to the standard decryption algorithm for various modulus sizes (e.g., $100, 200, 400, 600, 800; \cdots$ decimal digits) and 10 random plaintext per modulus. In your implementation, p and q should be random primes in the interval $(10^{s-1}, 10^s)$, when the modulus size is $2s$ decimal digits. The plaintext should be a random number in the range $0 < m \leq n-1$. You can take d to be a random prime number that does not divide $(p-1)(q-1)$, and then compute e such that $e \equiv d^{-1} \mod (p-1)(q-1)$.

(b) [6 marks] The receiver knows the private key, so he can precompute the following

$$d_1 \equiv d \mod (p-1) \qquad \text{and} \qquad d_2 \equiv d \mod (q-1).$$

He can also precompute m_1 and m_2 such that

$$m_1 q \equiv 1 \mod p \qquad \text{and} \qquad m_2 p \equiv 1 \mod q.$$

During the *online* phase, when he gets the ciphertext c, the receiver computes

$$x_1 \equiv c^{d_1} \mod p, \qquad x_2 \equiv c^{d_2} \mod q,$$
$$m \equiv (m_1 q x_1 + m_2 p x_2) \mod n.$$

Perform all the computations using the decryption algorithm outlined above, with the same choice of the parameters as in part (a) (3 marks). Compare the online phase of both decryption routines by plotting a graph of moduli size vs average decryption time (3 marks).

You can refer to the standard MAPLE commands given at the end of this assignment to answer this question.

3. [8 marks total] **GnuPG.**
A GnuPG public key for `jkupadhy@cs.uwaterloo.ca` is provided along with the assignment on the course website (jkupadhy.asc). Perform the following tasks. You can install GnuPG on your own computer, or use the version we have installed on the ugster machines.

   (a) [2 marks] Generate a GnuPG key pair for yourself. Use the RSA and RSA algorithm option, your real name, and an email address of your-userid@uwaterloo.ca. Export this key using ASCII armor into a file called **key.asc**. [Note: older versions of GnuPG might not have the RSA and RSA algorithm option, so check that the version you are using has this option. The ugster machines have a new enough version, but the student.cs machine does not.]

   (b) [2 marks] Use this key to sign (not local-sign) the jkupadhy@cs.uwaterloo.ca key. Its true fingerprint is: `86BB F73A 20E8 1304 8E18 6D83 DA04 A4D4 F2F7 C84E`. Export your signed version of the jkupadhy key into a file called **jkupadhy-signed.asc**; be sure to use ASCII armor. [Note: signing a key is not the same operation as signing a message.]

   (c) [2 marks] Create a message containing your userid and name. Sign it using the key you generated, and encrypt it to the jkupadhy key. You should do both the encryption and signature in a single operation. Make sure to use ASCII armor, and save the output in a file called **message.asc**.

   (d) [2 marks] Briefly explain the importance of fingerprints in GnuPG. In particular, explain how users should check fingerprints and what type of attacks are possible if users do not follow this procedure properly.

4. [6 marks total] **A Practical Attack.**
Suppose the CS458 course staff are testing a new "secure" mark submission system. Midterm and Final exam marks are encrypted using CBC mode by a TA before being sent to the mark database. To convince you that a secure process is being used, a TA demonstrates the process for a fake student the course staff use for testing purposes:

```
$echo aaaaaaaa 100 100 > message

$openssl enc -aes-128-cbc -iv 00000000000000000000000000000000
   -pass pass:password -in message -out ciphertext

$od -x ciphertext
0000000 6153 746c 6465 5f5f b935 2b69 2f53 2b0d
0000020 733f b288 72c1 26bf a340 2f3c f7b6 0959
0000040 8333 e650 277a 718b 540c 48a4 ee75 ecb7
0000060
```

This (ciphertext, IV) tuple is sent to the database to update the mark. Of course, the TA would never actually reveal the password used to the class. You can assume the key was read from a keyfile or other private source and not actually shown to the class. The TA knows that an IV is not meant to be secret, and does not bother to hide it.

The database receives the encrypted mark and can recover it by decrypting:

```
$openssl enc -d -aes-128-cbc -iv 00000000000000000000000000000000
   -pass pass:password -in ciphertext

aaaaaaaa 100 100
```

While answering student questions, the TA explains that each update is of the form [8 character userid][space][3 digit midterm mark][space][3 digit exam mark]. The database knows an update is valid if the userid is found in the database and the marks are between 000 and 100. Because AES is secure, no one but a TA with the password should be able to construct a valid mark submission.

This question explore the security of the TAs' approach to encryption. You might find Wikipedia's article on the CBC mode of operation useful for this question: `https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation#Cipher-block_chaining_.28CBC.29`

(a) [3 marks] Demonstrate that the above system is not secure. Specifically, provide an IV that, when sent with the ciphertext above to the database, results in a mark update for your personal userid instead of user "aaaaaaaa" (2 marks). Give the technical reason why your attack works (1 marks).

(b) [3 marks] Which CIA principle(s) is/are violated (1 marks)? How would you fix it (2 marks)?

You can assume that the message fits entirely within the first block of the plaintext.

Programming Question [20 marks]

A hospital in Waterloo wants to make a database of infectious diseases in Waterloo Region available to researchers. The corresponding file, **Infectious_Disease_Record.csv**, is provided on the course website along with the assignment. For every person living in Waterloo Region, this database contains at most one record. A record lists the person's name, gender, telephone number, disease diagnosis, postal code, and date of birth. Understanding that it is important to preserve the privacy of these records, the hospital hires you as a consultant to anonymize the database. Your job is to anonymize the record based on k-anonymity. In general, the problem of k-anonymizing a database while maximizing the amount of remaining useful information is NP-hard; therefore, we only ask you to implement a very specific anonymization strategy, consisting of four stages. (We will provide bonus marks for a polynomial-time solution to the aforementioned NP-hard problem; we'll also help you claim a million dollar prize from the Clay Mathematics Institute.)

- In each record, the name and phone number are each replaced with an asterisk (*).
- Based on a person's gender, postal code, and date of birth, it might be possible to re-identify the person, e.g., by correlating this information with a public polling list from a recent provincial election. k-anonymity can help here. Assume that there is a set of records that all have the same gender, postal code, and date of birth. If there are fewer than k records in this set, the day in the date of birth is replaced with an asterisk for each record in this set. Repeat this process for each possible set.
- The second stage is repeated, but this time, the month in the date of birth is replaced with an asterisk for all sets of records that still have fewer than k records.
- The second stage is repeated, but this time, the last three characters in the postal code are replaced with an asterisk for all sets of records that still have fewer than k records.

You should hand in a file, **anonymized.csv**, containing all the anonymized records. Furthermore, you should hand in a second file, **non-kanonymized.csv**, containing all the records from the original database for which k-anonymity does not hold after the anonymization. When generating the two files, set $k = 6$, include column labels (as in the examples), and do not reorder records. Write a program to help you with the anonymization. You can use any programming language you like, but the program will need to compile (if applicable) and run on an ugster machine. You will need to submit your program (including Makefile), as well as the two files mentioned above. To get full marks, the total running time of your program should be shorter than one minute on an ugster machine.

Note: We will test your program on input databases other than the one provided to you!

Example with $k = 2$
For example, assume the database consists of the following four records (plus header):

Name, Gender, Date of birth, Disease, Postal code, Telephone
Alice Alisson, F, 1911-11-1, Yellow fever, N2L 3G1, 519 123 3435
Bob Bobson, M, 1945-12-4, Mumps, N2L 4F2, 519 537 8756
Carol Carson, F, 1911-11-8, Conjunctivitis, N2L 3G1,519 635 5285
Dave Davidson, M,1978-4-3, Hamburger disease, N2L 3G0, 519 274 4345

Then **anonymized.csv** would look as follows ($k = 2$):

Name, Gender, Date of birth, Disease, Postal Code, Telephone
$*$, F, 1911-11-$*$, Yellow fever, N2L 3G1, $*$
$*$, M, 1945-$*$-$*$, Mumps, N2L $*$, $*$
$*$, F, 1911-11-$*$, Conjunctivitis, N2L 3G1,$*$
$*$, M,1978-$*$-$*$, Hamburger disease, N2L $*$, $*$

Note that the resulting database is not strictly k-anonymous. Namely, there is only one record with year 1945. Same for 1978. Therefore, **non-kanonymized.csv** would look as follows:

Name, Gender, Date of birth, Postal code, Telephone, Disease
Bob Bobson, M, 1945-12-4, Mumps, N2L 4F2, 519 537 8756
Dave Davidson, M,1978-4-3, Hamburger disease, N2L 3G0, 519 274 4345

To simplify correctness checking of your program, we make available the SHA1 checksums of the database after each anonymization stage mentioned above. You can compute the SHA1 checksum of your database using the `sha1sum` program. Make sure to have a newline character ('\n'), and no carriage return character ('\r'), at the end of each line. If you have difficulties getting the checksums right, it might be due to a problem with whitespace. To debug this problem, insert an initial dummy stage in your program, which simply reads the original database, processes (but does not filter) it, and writes it to an output file. Use the `diff` program or a hex editor to compare the output file to the original database.

- Original file: `726cff82a1f877f9e5e39b73c8769a3b84b197e9`
- After first stage: `214dbf6805d7c7267a4bf21cdf85792cc222db17`
- After second stage: `c799bd77d1b6c06a57b36d299da61c4ef7ceac02`
- After third stage: `a340246ea1ef07ba93f8df3d8ce528cf6df48a82`

- After fourth stage (this should be SHA1 checksum of **anonymized.csv**): `196c5c8dd3459ee3987aef90f9df1bf0806a52df`
- The `sha1sum` for the **non-kanonymized.csv** should be `5ecf44fca2ae54582e6e81c728c1d42b379d559c`

Your program will be marked as follows:

- 10 marks for submitting correct **anonymized.csv** and **non-kanonymized.csv** files. Your code must be able to correctly generate these files when provided with the sample input.
- 8 marks in total: 2 marks each for correctly implementing each of the 4 anonymization steps according to our additional test data.
- 2 marks for correctly generating **non-kanonymized.csv** using our additional test data.

Your program should read the input file **Infectious_Disease_Record.csv** (not .zip) from the current working directory, and should create the following output files:

- **anonymized.csv.1**– Output after first stage of anonymization
- **anonymized.csv.2**– Output after second stage of anonymization
- **anonymized.csv.3**– Output after third stage of anonymization
- **anonymized.csv**– Output after final stage of anonymization
- **non-kanonymized.csv**– The complete records for which k-anonymity does not hold

You should use the provided test data and SHA1 sums to ensure that your output is formatted correctly at each step. All output files are expected to contain the same column labels as the provided input file. Any additional test data will be identical in format to the provided sample data.

What to Hand in

Using the "*submit*" facility on the student.cs machines (**not** the ugster machines or the UML virtual environment), hand in the following files:

a3.pdf: A PDF file containing your answers for all written questions. It must contain, at the top of the first page, your name, UW userid, and student number. -3 **marks if it does not**! Be sure to "embed all fonts" into your PDF file. Some students' files were unreadable in the past; if we cannot read it, we cannot mark it.

RSA*: The source code to your program from Question 2 and the graph mentioned in part (b).

key.asc, jkupadhy-signed.asc, message.asc: the ASCII-armored files from Question 3.

anonymized.csv: Your anonymized records from the programming question.

non-kanonymized.csv: Your non-anonymized records from the programming question.

kanon*: The source code to your program from the programming question. All of your source code filenames should start with "kanon", but can be anything (with any extension) after that. This allows you to write your program in any language.

Makefile: The Makefile used to compile (if applicable) and run your program from the programming question. It must have a target "make run", which will compile (if applicable) and run your program.

Useful MAPLE commands

The list is not exhaustive. MAPLE provides many other commands for the same effect. However, for large number arithmetic, following commands are better to use.

nextprime(n) outputs the next prime number after n.

rand(r) Outputs a random number. With an integer range as an argument, the call rand(a..b) returns a procedure which, when called, generates random integers in the range $[a, b]$. With a single integer as an argument, the call rand(n) gives a random number less than n.

isprime(n) returns a boolean value depending on whether or not n is prime or not.

Power(a,b) mod c outputs the value $c \equiv a^b \mod c$. Another command that is equivalent to Power command is $a\&\hat{}b \mod n$

Inverse of a modulus b can be computed either by the Power command or by $\frac{1}{a} \mod b$, if it exists.

If you want to find the time elapsed between two or more lines of codes, MAPLE has the following set of command that are used in conjunction.

```
st:=time();
 \\ lines of MAPLE code
elapsed_time:=(time()- st)*sec;
```

Another way to generate random number is as follows.

```
with(RandomTools);
seed := any desired value;
SetState( state = seed );
lower_bound := any desired value;
upper_bound := any desired value;
Generate(integer(range = lower_bound..upperbound));
```

You might need to include the package $numtheory$ for many of the above commands. The way to load a package package in MAPLE, you need to type the command $with$(package).